REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-02-

0077

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 9/30/2001	3. REPORT TYPE AND DA... COVERED Final report March 15, 1998 – June 1, 2001

4. TITLE AND SUBTITLE Semantic Issues for Component Technologies	5. FUNDING NUMBERS AFOSR G4514209-12 F49620-98-1-0360
6. AUTHOR(S) Beverly A. Sanders	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer and Information Science and Engineering P.O. Box 116120 University of Florida Gainesville, FL 32611-6120	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR 801 N. Randolph Street, Room 732 Arlington, VA 22203-1977	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	20020315 085
12a. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A Approved for Public Release Distribution Unlimited AIR FO... NO... H... L...	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

The availability of components that can be assembled together to build a customized system promises to decrease software development costs and, one hopes, also increase the reliability of the resulting system. Clearly, this promise can only be realized when a system designer has adequate information about the components that will be used to predict how systems containing them will behave. This project explored approaches to specifying components using formal techniques that allow a component developer to verify that the component itself satisfies its specification, and also allow the system builder who uses the component to reason about the behavior of systems containing that component.

A general theory guarantees properties for components was developed. A guarantees property of a component describes the behavior of systems containing that component. In order to use the general theory, it must be instantiated as and extension of a more complete programming logic. Case studies were performed with two programming logics: UNITY, and CTL. In the latter case, model checking tools for mechanized reasoning about closed systems specified in CTL were used along with assertional techniques to reason about composite systems.

14. SUBJECT TERMS Composition, program verification, temporal logic	15. NUMBER OF PAGES 15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Final Report: Semantic Issues for Component Technologies

Beverly A. Sanders
Department of Computer and Information Science
University of Florida Gainesville, FL 32611-6120
AFOSR Grant 4514209-12

September, 2001

1 Status of Effort

The availability of components that can be assembled together to build a customized system promises to decrease software development costs and, one hopes, also increase the reliability of the resulting system. Clearly, this promise can only be realized when a system designer has adequate information about the components that will be used to predict how systems containing them will behave. This project explored approaches to specifying components using formal techniques that allow a component developer to verify that the component itself satisfies its specification, and also allow the system builder who uses the component to reason about the behavior of systems containing that component.

A general theory guarantees properties for components was developed. A guarantees property of a component describes the behavior of systems containing that component. In order to use the general theory, it must be instantiated as and extension of a more complete programming logic. Case studies were performed with two programming logics: UNITY, and CTL. In the latter case, model checking tools for mechanized reasoning about closed systems specified in CTL were used along with assertional techniques to reason about composite systems.

2 Accomplishments

2.1 Introduction

Suppose that a component, call it F, satisfies some specification, call it *SpecF*, and another component G satisfies a specification *SpecG*. A system builder hoping to utilize these components would like to answer "What can one then say about $F \circ G$, a system built by combining, or composing, F and G?".

As a simple example that illustrates the generality of the issue: suppose *SpecF* is "F weighs less then 10 grams" and *SpecG* is "G weighs less than 15 grams." We can immediately conclude that $F \circ G$ weighs less than 25 grams. This particular form of specification, namely giving the weight of some physical component, admits simple reasoning in composed systems since weight is additive.

An example more typical of the intended domain of exploration, concurrent software systems, is the following: Suppose component F controls a resource and satisfies the specification: "if F holds the resource and it is requested by a client, then the client with the earliest request will eventually be granted the resource." Suppose the clients all satisfy the specification: "if a client G holds the resource, it will eventually return it to the the controller." Can one then conclude that in a system composed of the controller and clients that "all requests for the resource will eventually be granted"?

Unfortunately, these specifications cannot be composed as simply as the component weight example. Depending on how the components interact in ways not specified, the desired conclusion may or may not hold. A goal of this project was to increase the fundamental understanding of composition and develop techniques for sound reasoning about composition, both as a general concept, and in the specific context of concurrent and distributed programming.

This work will contribute to the ability of the Air Force to leverage the development effort of software components by allowing them them to be used in multiple systems. By understanding the issues affecting the behavior of systems in composition, component developers can design components that are easier to reason about and use. More effort can be placed in the validation of the components, since this work is "reused" when the components are reused, provided the component specifications are developed with the needs of composition in mind. With sound techniques for reasoning about components, the confidence in systems that use them can be increased.

The approach taken in this project can be summarized as follows:

- Look for simple and general notions of composition.

- Identify classes of properties that compose well

- Develop a calculus of composition

- Instantiate the general theory as an extension of a specific programming logic, in this case UNITY and CTL.

- Since CTL is well supported by tools for model checking, the use of model checking for reasoning about program composition was investigated.

2.2 Technical Results

2.2.1 Definitions

Let $\mathcal{PG}$ be a set of systems with well-defined semantics. A *property* is a predicate (boolean valued function) on $\mathcal{PG}$. Since properties are predicates, the set of properties is closed under logical negation, conjunction, and disjunction. We use symbols F, G and H for systems, X, Y, and Z for properties. So, $Z.H$ is function Z applied to argument H, and therefore, $Z.H$ is the boolean: property Z holds for system H.

Composition is denoted with a composition operator $\circ$ of type $\mathcal{PG}$ x $\mathcal{PG} \rightarrow \mathcal{PG}$. We require that $\circ$ is associative with identity SKIP, where SKIP $\in \mathcal{PG}$.

and if a composed system is in $\mathcal{PG}$ then all its components are also in $\mathcal{PG}$:

$$F \circ G \in \mathcal{PG} \Rightarrow F \in \mathcal{PG} \wedge G \in \mathcal{PG}.$$

We begin by assuming that $\circ$ is symmetric (commutative). The set $\mathcal{PG}$ need not be closed under composition: It is possible that $F \circ G$ is not in $\mathcal{PG}$ even though F and G are.

The function *component* from systems to properties is defined as follows: For a system F, *component.F* is a property that holds for all systems that can have F as a component and only such programs.

$$\mathit{component}.F.H \equiv (\exists G : F \circ G \in \mathcal{PG} \ : F \circ G = H)$$

From the definition of *component*, if G is a component of H and H is a component of F then G is also a component of F:

$$\mathit{component}.G.H \wedge \mathit{component}.H.F \Rightarrow \mathit{component}.G.F$$

Since SKIP can be composed with any program, and F composed with SKIP is F itself, F is a component of itself:

$$(\forall F : F \in \mathcal{PG} \ : \mathit{component}.F.F)$$

and SKIP is a component of all systems:

$$[\mathit{component}.SKIP]$$

2.2.2 Classes of easily composable properties

Two classes of properties of interest were identified:

- **Existential**. A property is existential if holding for a single component is sufficient for the property to hold for any system containing that component.

  More formally, a property X is an *existential property* exactly when for all $\mathcal{G}$

  $$(\exists G : G \in \mathcal{G} : X.G) \;\Rightarrow\; X.(\circ G : G \in \mathcal{G} : G) \qquad (1)$$

  An example is a property that states that "eventually the value of variable x will change". Assuming fairness, it only requires one component to guarantee this for it to hold in the entire system

- **Universal** Universal properties hold for a system if they hold for each component in the system. A property X is an *universal property* exactly when for all $\mathcal{G}$,

  $$(\forall G : G \in \mathcal{G} : X.G) \;\Rightarrow\; X.(\circ G : G \in \mathcal{G} : G) \qquad (2)$$

  An example is a property that states that "x never decreases". This will be the case if no component ever decreases the value of x, (assuming an underlying model that allows each component to read and modify x atomically.)

Existential and universal properties are very convenient because they allow reasoning about a system by reasoning about the individual components. Clearly, not all properties of interest are existential or universal. Among other issues, a way to talk about constraints on the environment in which a component is deployed is needed. To do this the notion of a guarantees property, based on ideas originally proposed by Pnueli [Pnu84] was define. A **guarantees** property (X *guarantees* Y) is a property of a system F if and only if all systems that have F as a component and have X as a property also have Y as a property. We refer to properties of the form (X *guarantees* Y)

as *guarantees* properties. A *guarantees* property is a higher-order property since it depends on the entire set of systems. More formally:

$$(X \textit{ guarantees } Y).F = (\forall H : \textit{component}.F.H : X.H \Rightarrow Y.H) \qquad (3)$$

An example of a guarantees property is "the value of x is nondecreasing" *guarantees* eventually x will be at least 10. It would be expected that this property would hold for any component, say F with the existential property that "eventually the value of x will change". As discussed above, the left side of the guarantees property is a universal property, thus could be checked by checking each component of the system individually. The guarantees property encapsulates the inductive reasoning needed to conclude the right side of the guarantees property so that it can be essentially reused every time the component F is deployed.

In contrast to a common interpretation of rely/guarantee specifications, the meaning of $(X \textit{ guarantees } Y).F$ is *not* that if the *environment* H of F has property X then the composed system $F \circ H$ has property Y. This interpretation leads to difficulties when viewing a composed system as a component in a larger system.

Guarantees means that if the *composed* system $F \circ H$ has property X then the composed system $F \circ H$ has property Y. Since both the left and right hand sides of the guarantees refer to properties of the *same* system, from the associativity of composition, guarantees properties are existential. Thus if $X \textit{guarantees } Y.F$ holds for some component F, then $X \textit{guarantees } Y.F \circ G$ Further, *guarantees* satisfies many of the rules of implication. It is disjunctive in the left operand, conjunctive the right operand, and satisfies a contrapositive property. A useful theorem about guarantees that was used implicitly in the example above is Given $X \wedge Y \Rightarrow Z$ where X is universal and Y is existential, is $Y.F \Rightarrow (X \textit{guarantees } Z).F$.

An additional property *env* has been studied. *env.P.F* means that P must hold for any component that can be composed with F. Typically, an env property P is most useful if it can be verified during compilation or linking. Knowledge of the *env* property can be used to weaken the left hand side of guarantees properties, essentially transferring some of the proof burden from the programmer.

To composition theory described so far can be summarized as comprising

- A general and simple notion of composition

- Identification of important classes of properties: existential, universal, guarantees, and env.

- Rules for manipulating properties

2.3 Specializing the compositional theory

The work involved in proving that a system satisfies some properties involves the following:

- The component developers prove that the components satisfy their guarantees and other properties. Although this may be difficult, it is only done once.

- The systems developer who uses a component must prove that the system satisfies the left had side of the guarantees properties. Since this must be done each time the component is used in a system it is desirable that this be easy. It helps if the properties on the left side are universal or existential so that they may be checked separately for each component.

In order to prove that a component satisfies a specification, a model-specific theory is needed. In this project, two different programming models were instantiated.

2.3.1 UNITY

UNITY is a programming logic originally described in [CM88] with later versions in [Mis95a, Mis95b]. Although quite simple, Chandy and Misra, and others have demonstrated that many interesting programs can be specified in the model.

Programs, which are considered to generate infinite computations, are specified with 3 basic properties that form a fragment of linear time temporal logic.

- p *next* q means that $p \Rightarrow q$, and that during the entire computation, if p holds at some point, then q holds after the next step.

- *transient* p means that, if p holds at any point in the computation, eventually p will become false. Alternatively, *transient* p means that p

cannot remain true forever. Transient properties depend on an implicit fairness assumption.

- $p \leadsto q$ means that during the entire computation, if p holds at some point, then eventually q will hold.

A study of these properties revealed that *next* properties are universal, *transient* properties are existential. *leads – to* properties are neither, but a new axiomatization was developed that allows all *leads – to* properties to be derived from a set of *next* and *transient* properties.

These rules are as follows:

- Transient rule

$$\frac{transient\ p}{p \leadsto \neg p} \tag{4}$$

- Progress, safety, progress

$$\frac{p\ next\ q,\ r \leadsto s}{p \wedge r \leadsto (q \wedge r) \vee (\neg r \wedge s)} \tag{5}$$

- Disjunction

$$\frac{p_i\ next\ q}{(\vee i : p_i) \leadsto q} \tag{6}$$

- Transitivity

$$\frac{p \leadsto q,\ q \leadsto r}{p \leadsto r} \tag{7}$$

Additional proof rules, such as an induction rule, can be derived from these properties.

The importance of this rule is that when one wants a leads-to property on the right side of a guarantees property, the transient and next properties used in the hypothesis of the transient and progress, safety, progress rules become the properties on the left side of the guarantees property. In addition, any transient property satisfied by the component can be eliminated from the list of left-hand side properties. The remaining properties are either existential or universal. The user of the component need not perform a complicated progress proof, but only needs to prove a set of simpler next, and possibly, transient properties. Here, the guarantees property encapsulates difficult reasoning so that it only need be done once.

2.3.2 CTL

The theory of composition was also studied in the context of CTL. CTL is a branching time temporal logic. Among other reasons, it is of interest because of the availability of tools to verify finite state systems with properties specified in this logic using a technique called model checking [CGP99]. The model checker takes a description of a finite state system and a property, and either determines that all possible computations of the finite state system satisfy the property, or shows a counterexample. In contrast to theorem proving, model checkers do not (in principle) require cleverness or help from the user, thus have the potential to become more widely used than many other approaches to formal verification.

One of the major limiting factors in CTL model checking in practice is the state explosion problem, and this has motivated most of the work on using composition with CTL [BCC98, GL94]. By decomposing a system into parts that can be checked separately, larger systems can be handled. Much of the work also considers synchronous composition, the appropriate type of composition for hardware. This project considered open systems where the components are combined with asynchronous composition. Other work on model checking for open systems [KV97] has indicated that certain formulations of using model checking for components is intractable in general.

Rather than attempting to model check guarantees properties directly, the approach in this project was to combine model checking with deduction. Let P be a program property and f a component, the notation $P.f$ means that f satisfies property P. Also, let c_i be some component, the expansion of this component, denoted by $\overline{c}_i$, is a program that behaves as c_i but it has some additional variables[1]. These additional variables might have any initial value that does not change during the execution of $\overline{c}_i$. The propose validation process is as follows: Each component c_i has some properties A_i, E_i such that $A_i.c_i \implies \overline{A}.\overline{c}_i$ and $e_i.c_i \implies \overline{E}_i.\overline{c}_i$. $\overline{A}$ is universal and $\overline{E}_1 \ldots \overline{E}_n$ is an existential property. One can then conclude that the entire system (i.e., the composition of all the c_i's) satisfies $\overline{A} \wedge (\forall i :: \overline{E}_i)$. Since many important properties are neither existential nor universal guarantees properties of some components are used to prove these remaining properties. Suppose there is a component c_j such that $(F \textbf{ guarantees } G).c_j).\overline{c}_j$. Then, the entire system satisfies $\overline{G}$ if $\overline{A} \wedge (\forall i :: \overline{E}_i) \implies \overline{F}$. All the properties A_is and E_is should be verifiable using model checking only. A set of theorems

[1]The variables of the entire system that are not visible to this component

might also be published by the developer of the components to facilitate the proofs. Any proofs done by the component developer can be reused. Several case studies are described in [AG01, AS02, AS].

2.3.3 Discussion

In both UNITY and CTL, the treatment of initial conditions must be done very carefully. In a closed system, the initial conditions may be used to conclude that some states are unreachable. These states can then be ignored. When a component is placed in a larger system, however, interactions between components may cause some of these states to become reachable, and the unreachable states can not longer be ignored unless deliberate steps are taken to ensure that they will remain unreachable in the composition. This may involve introducing guarantees properties, or placing constraints on the composition.

A fairness assumption states that each component will be given a chance to execute. (There are several technical definitions of fairness that one could consider.) Fairness assumptions are essential to the identification of any progress properties as existential. In UNITY, fairness is an implicit part of the underlying programming model. In CTL, they are not. We found that, in contrast to the usual approach of considering fairness as part of the system, it is more convenient to consider both fairness and initial conditions as part of a property.

For UNITY logic, a very strong compositional result was obtained: All UNITY properties are either existential, universal, or follow from properties that are existential or universal. The proof of a property generates the left side of a guarantees property as a byproduct of the proof process. In CTL, some properties were identified as existential and universal and theorems developed. Since CTL is a much more expressive logic than UNITY, the compositional results are not as strong as in UNITY and are available for only a subset of CTL properties. Another factor is that in CTL, proofs are generally not constructed, but model checking is used instead. The value of these results depend on how useful they are in practice. Several case studies showed that the partial results were adequate for at least some interesting problems. Discovering the compositional characteristics of other property patterns remains an important problem for future work. The work on CTL is described in detail in [AG01] a PhD thesis by Hector Andrade-Gomez, and [AS02, AS].

2.4 Related work: A Pattern Language for Parallel Programming

At the suggestion of the program manager, this project is briefly described here. This work was funded by the National Science Foundation and Intel with collaborators Berna L. Massingill and Timothy G. Mattson.

A design pattern is a description of a high quality solution to a frequently recurring problem which presented in a prescribed format in order to help both the reader and the pattern writer focus on the essential aspects of the problem. A pattern language is a set of patterns that could used together, and which are organized into a structured hierarchical catalog. The pattern language for parallel application programming presents a collection of patterns that guide a programmer from the initial stages of parallel program design, such as finding the concurrency in a problem, to identifying the algorithm structure to be used along with patterns for supporting structures. The pattern language is described in [MMS99, MMS00b, MMS00a, MMS01] can be viewed at `www.cise.ufl.edu/research/ParallelPatterns`.

The experience shows that a pattern language can be an effective way to organize expertise in a given area. The important characteristics are

- Each pattern has a name, providing a common vocabulary for solutions
- Each pattern contains a concise description of the problem it will solve, the context where the solution is applicable, and the main tradeoffs that must be made.
- Patterns often contain references to related patterns, and to patterns that are likely to be useful in the next step of the design process.

Patterns may prove to be a useful way of organizing knowledge about composition as well.

References

[AG01] Hector A Andrade-Gòmez. *Model Checking for Open Systems: A Compositional Approach to Software Verification.* PhD thesis, University of Florida, 2001. http://etd.fcla.edu/etc/uf/2001/ank6403/thesis.pdf.

[AS] Hector A. Andrade and Beverly A. Sanders. Model checking for open systems. submitted for publication.

[AS02] Hector A. Andrade and Beverly A. Sanders. An approach to compositional model checking. In *Proceedings of Formal Methods for Parallel Programming: Theory and Applications*, Fort Lauderdale, 2002. IPDPS2002, IEEE Press. to appear.

[BCC98] Sergey Berezein, Sergio Campos, and E.M. Clarke. Compositional reasoning in model checking. In *Proceedings of the Workshop COMPOS'97*, February 1998.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT press, 1999.

[CM88] K. Many Chandy and R. Misra. *Parallel Programing a Foundation.* Addison Wesley, 1988.

[GL94] Orna Grumberg and David Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[KV97] Kupferman and M. Y. Vardi. Module checking revisited. In *Proc. of the 9th conference on Computer-Aided Verification (CAV97)*, pages 36–47, 1997.

[Mis95a] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[Mis95b] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[MMS99] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99)*, 1999. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS00a] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A pattern language for parallel application programming. In *Proceedings of the Sixth International Euro-Par Conference (Euro-Par 2000)*, 2000.

[MMS00b] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for finding concurrency for parallel application programs. In *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP'00)*, August 2000. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[MMS01] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Parallel programming with a pattern language. *International Journal on Software Tools for Technology Transfer*, 3(2), 2001.

[Pnu84] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1984.

3 Personnel Supported

- Beverly A. Sanders, faculty
- Hector Andrade-Gomez, graduate research assistant
- Rohit Mavarapu, graduate research assistant

4 Publications

- Hector A Andrade-Gòmez. *Model Checking for Open Systems: A Compositional Approach to Software Verification.* PhD thesis, University of Florida, 2001. http://etd.fcla.edu/etc/uf/2001/ank6403/thesis.pdf.
- Hector A. Andrade and Beverly A. Sanders. Model checking for open systems. submitted for publication to Formal Aspects of Computing.
- Hector A. Andrade and Beverly A. Sanders. An approach to compositional model checking. In *Proceedings of Formal Methods for Parallel Programming: Theory and Applications*, Fort Lauderdale, 2002. IPDPS2002, IEEE Press. to appear.

- Beverly A. Sanders. Using atomic await commands to develop concurrent programs in Java. *Software-Concepts and Tools*, 19, 2000.

- Beverly A. Sanders. The shortest path in parallel. *Information Processing Letters*, 77, 2001.

- Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99)*, 1999. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

- Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A pattern language for parallel application programming. In *Proceedings of the Sixth International Euro-Par Conference (Euro-Par 2000)*, 2000.

- Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for finding concurrency for parallel application programs. In *Proceedings of the Seventh Pattern Languages of Programs Workshop (PLoP'00)*, August 2000. See also our Web site at `http://www.cise.ufl.edu/research/ParallelPatterns`.

- Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Parallel programming with a pattern language. *International Journal on Software Tools for Technology Transfer*, 3(2), 2001.

5 Interactions

5.1 Participation/presentations

The PI has organized the following workshops during the term of this grant

- International Workshop on Formal Methods for Parallel Programming: Theory and Applications, 2001. In conjunction with IPDPS 2001, San Francisco, CA.

- International Workshop on Formal Methods for Parallel Programming: Theory and Applications, 2001. In conjunction with IPDPS 2000, Cancun, MX

- International Workshop on Formal Methods for Parallel Programming: Theory and Applications, 2001. In conjunction with IPDPS 1999, Orlando, FL.

The PI has participated (or was an author of a paper presented by a coauthor) in the following conferences

- Europar 2000. Munich, 2000.
- Sixth Workshop on Pattern Languages of Programs (PLoP99), Urbana, 2001.
- Seventh Workshop on Pattern Languages of Programs (PLoP2K), Urbana, 2000.
- Symposium in Honor of Edsger W. Dijkstra. Austin, 2000.

5.2 Transitions

Dr. Laurence Paulson, of Cambridge University has mechanized our UNITY instantiation of the guarantees properties in the Isabelle theorem prover. This work not only has the potential to provide a useful tool for system developers who work with UNITY, but has also let to important insights about composition in strongly typed formalisms.

6 New Discoveries

None